two-phase commit

changelog

21 April 2022: rephrase naive distributed transaction exercises to avoid
using the confusing phrasing ‘agree’/‘do not agree’

last time

gRPC design choices

- expose local object to represent functions on server
- call method of local object → causes method call on server
- programming-language-neutral IDL
- arguments/return value always struct-like message

a model for network failure

- messages lost and reordered
- handle with acknowledgments, sequence numbers
- still can't be sure if message lost or ack lost
- e.g. RPC fails? may/may not have been done on server!

machine failure models

- fail-stop: machine stops responding (or we ignore it)
- Byzantine: machine can do anything
- reality: somewhere in between

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

two models of machine failure

fail-stop

failing machines stop responding/don't get messages

or one always detects they're broken and can ignore them

Byzantine failures

failing machines do the worst possible thing

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F + 1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F + 1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

distributed transaction problem

distributed transaction

two machines both agree to do something *or not do something*

even if *a machine fails*

primary goal: *consistent* state

secondary goal: do it if nothing breaks

distributed transaction example

course database across many machines

machine A and B: student records

machine C: course records

want to make sure machines agree to add students to course

no confusion about student is in course even if failures

"consistency"

okay to say "no" — if possible, can retry later

naive distributed transaction? (1)

machine A and B: student records; machine C: course records

any machine can be queried directly for info (e.g. by SIS web interface)

proposed add student to course procedure:

execute code on A or B where student is stored

tell C: add student to course

wait for response from C (if course full, return error)

locally: add student to course

exericse (1)

seperate student (local) + course (remote) records

tell remote: add student to course

then locally: add student to course

if no failures, which are possible to observe from third machine (that asks student/course machines for current records)?

A student record: in course; course record: not in course; but if double checking: both agree

B same as A, but if double-checking both *do not* agree

C student record: not in course; course record: in course; but if double checking: both agree

D same as C, but if double-checking both *do not* agree

exericse (2)

seperate student (local) + course (remote) records

tell remote: add student to course

then locally: add student to course

if machine power loss + restart, which are possible to observe from third machine (that asks student/course machines for current records)?

A student record: in course; course record: not in course; but if double checking: both agree

B same as A, but if double-checking both *do not* agree

C student record: not in course; course record: in course; but if double checking: both agree

D same as C, but if double-checking both *do not* agree

decentralized solution properties

each machine handles only its own data

- no sending everything through one machine (easy solution)

machines involved in transaction if and only if have relevant data

- change only to courses? don't tell student machines
- change to course + student A? don't tell machine with student B

make progress as long as relevant machines don't fail

- losing one of K student machines? still runs for 1 of K students

decentralized solution properties

each machine handles only its own data

- no sending everything through one machine (easy solution)

machines involved in transaction if and only if have relevant data

- change only to courses? don't tell student machines
- change to course + student A? don't tell machine with student B

make progress as long as relevant machines don't fail

- losing one of K student machines? still runs for 1 of K students

hope: scales to tens/hundreds of machines

- typical transaction: 1 to 3 machines?

two-phase commit

will look at solution that satisfies these propties

known as two-phase commit

name from two steps: figure out what to do, then do it

hint: similar idea to redo logging

record intended actions, then do them

persisting past failures

will still use presistent log on each machine

idea: machine remembers what it was doing on failure

doesn't store data of other machines

...just some identifier/contact info for the transaction

two-phase commit: roles

one machine = *coordinator*

other machines are *workers*

- common implementation: one physical machine runs coordinator+one worker

two-phase commit: roles

one machine = *coordinator*

other machines are *workers*

- common implementation: one physical machine runs coordinator+one worker

key rule: abort (don't change anything) if anyone decides to abort

two-phase commit: roles

one machine = *coordinator*

other machines are *workers*

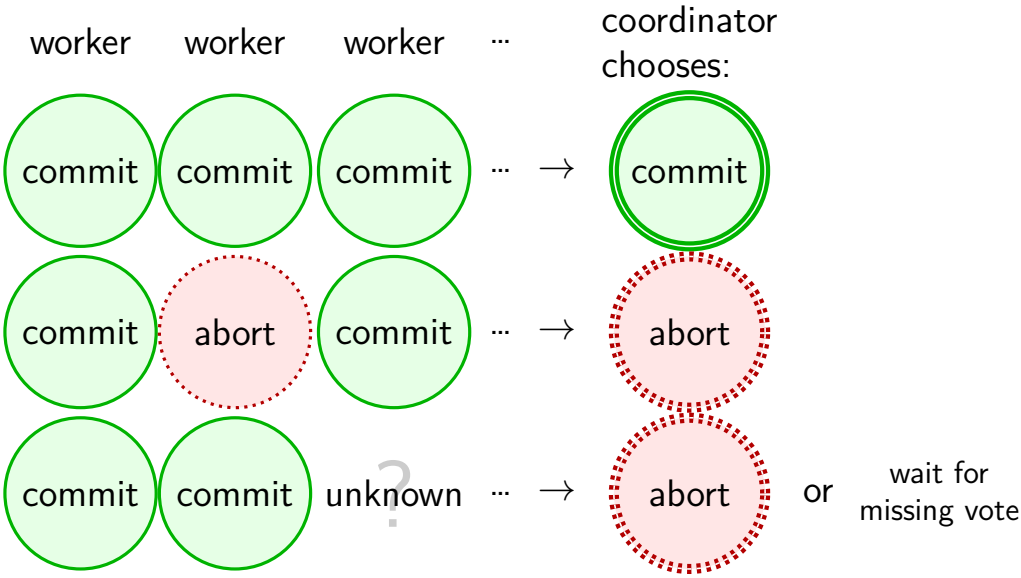
common implementation: one physical machine runs coordinator+one worker

key rule: abort (don't change anything) if anyone decides to abort

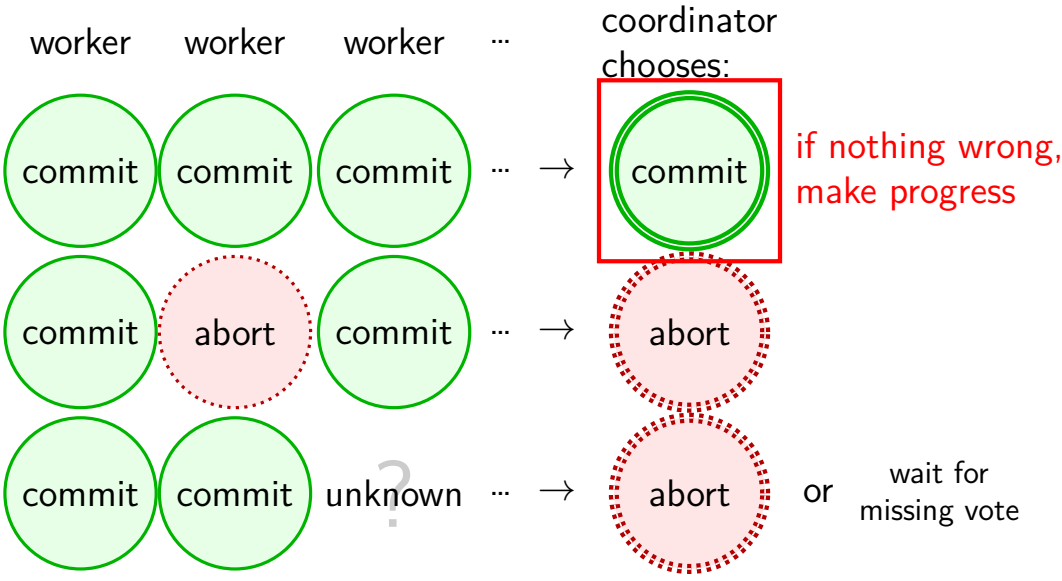
coordinator collects workers' vote: will they abort?

coordinator makes final decision using votes

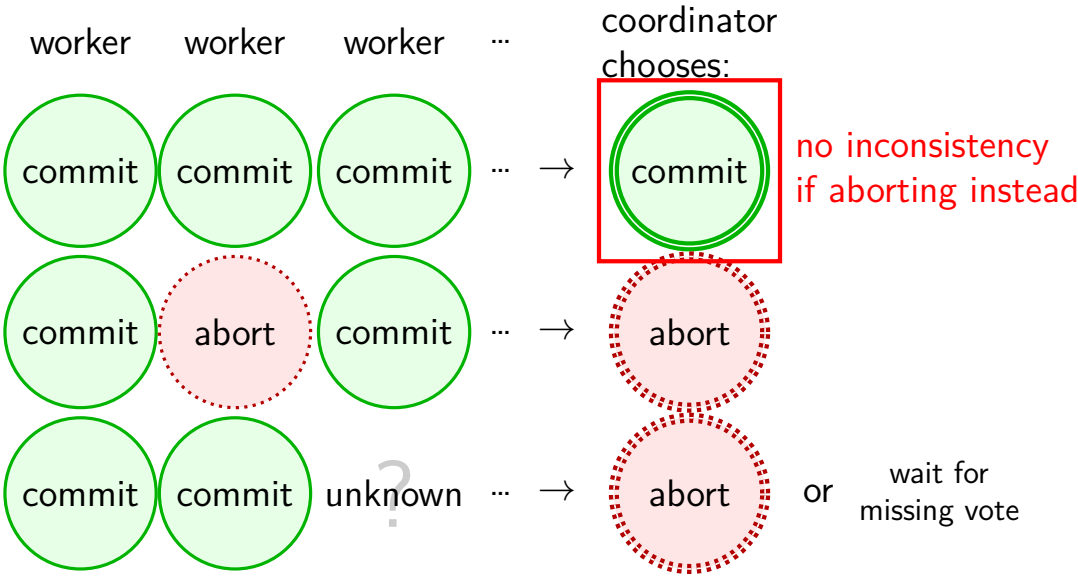
two-phase commit: voting



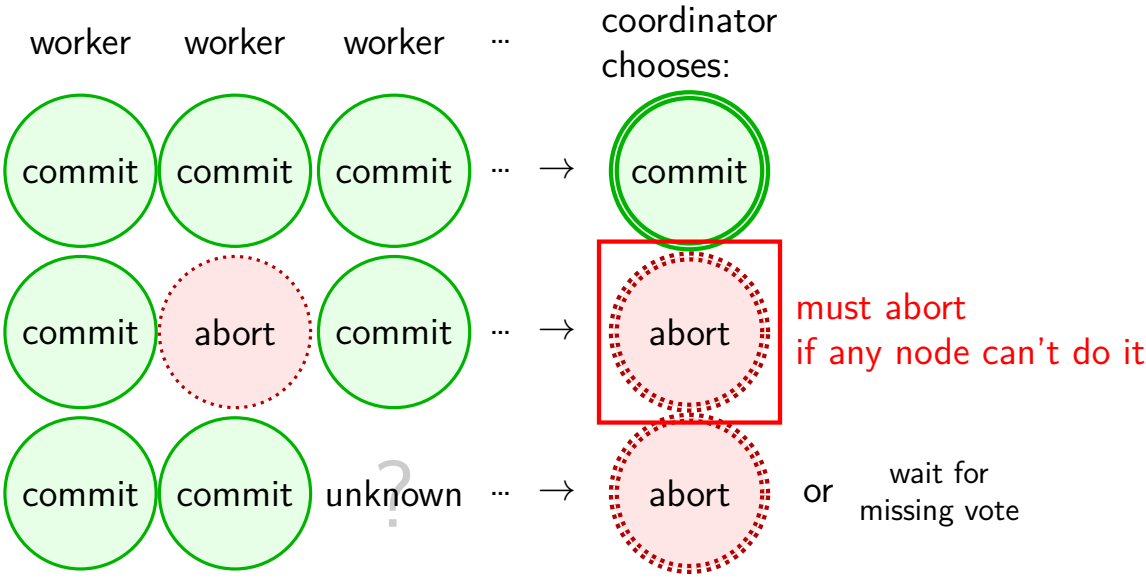
two-phase commit: voting



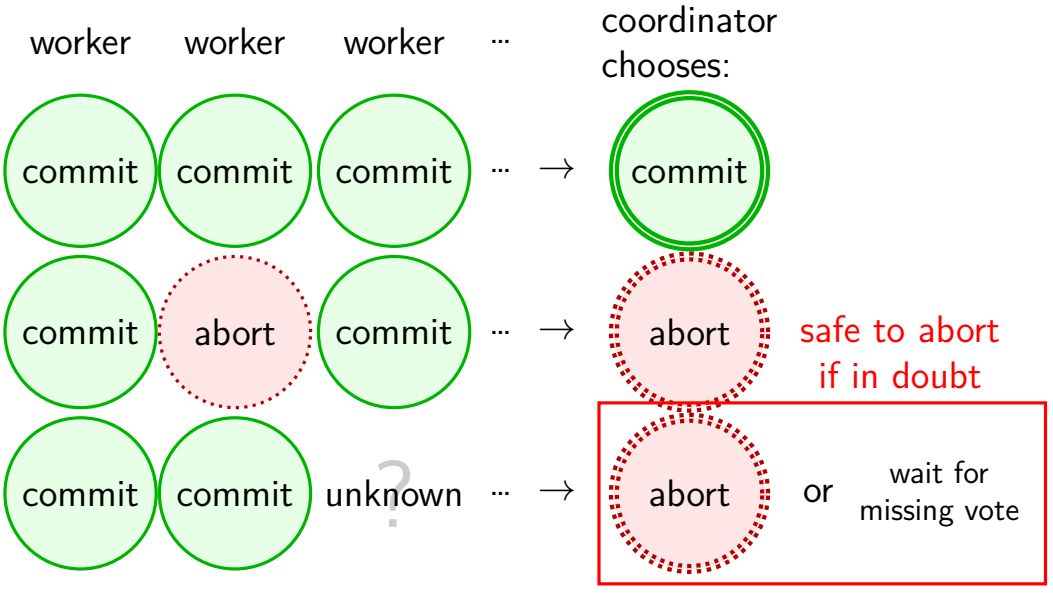
two-phase commit: voting



two-phase commit: voting



two-phase commit: voting



aside: why abort? (1)

why might worker want to abort?

simpliest example: operation not possible

course full

course doesn't exist on worker

worker out of disk space

...

aside: why abort? (2)

why might worker want to abort?

sublte issue: conflict with other tranaction; example:

aside: why abort? (2)

why might worker want to abort?

sublte issue: conflict with other tranaction; example:

transaction 1: worker agreed to add student X to course A

...but still waiting to confirm that this will happen

tranasction 2: worker asked to add student Y to course A

if course would be full after transaction 1, worker can't say 'yes'

aside: why abort? (2)

why might worker want to abort?

sublte issue: conflict with other tranaction; example:

transaction 1: worker agreed to add student X to course A

...but still waiting to confirm that this will happen

tranasction 2: worker asked to add student Y to course A

if course would be full after transaction 1, worker can't say 'yes'

option one: worker aborts, says "not now"

option two: worker delays response for transaction 2 until ready

aside: consistency and reads

don't want to allow reads of values that "in flux"

typical solution: reads need transaction, too

even though they don't change anything

assignment: workers have "unavailable" flag

two-phase commit: no take-backs

once worker agrees not to abort, it cannot change its mind

once coordinator makes decision, it cannot change its mind

two-phase commit: no take-backs

once worker agrees not to abort, it cannot change its mind

once coordinator makes decision, it cannot change its mind

both cases: need to remember decision after power loss, crash, etc.

solution: write decision down in log before acting on it

two-phase commit: phases

phase 1: *preparing*

workers tell coordinator their votes: agree to commit/abort

phase 2: *finishing*

coordinator gathers votes, decides and tells everyone the outcome

preparing

agree to commit

promise: "I will accept this transaction"
promise recorded in the machine log in case it crashes

agree to abort

promise: "I will **not** accept this transaction"
promise recorded in the machine log in case it crashes

never ever take back agreement!

preparing

agree to commit

promise: "I will accept this transaction"

promise recorded in the machine log in case it crashes

agree to abort

promise: "I will **not** accept this transaction"

promise recorded in the machine log in case it crashes

never ever take back agreement!

to keep promise: can't allow interfering operations

e.g. agree to add student to class $\rightarrow$ reserve seat in class

(even though student might not be added b/c of other machines)

coordinator decision

coordinator can't take back global decision

must record in presistent log to ensure not forgotten

coordinator decision

coordinator can't take back global decision

must record in presistent log to ensure not forgotten

coordinator fails without logged decision? collect votes again

finishing

coordinator says commit $\rightarrow$ commit transaction

- worker applies transcation (e.g. record student is in class)

coordinator (or anyone) says abort $\rightarrow$ abort transaction

- worker never ever applies transaction
- still want to do operation? make a new transaction

finishing

coordinator says commit → commit transaction

- worker applies transcation (e.g. record student is in class)

coordinator (or anyone) says abort → abort transaction

- worker never ever applies transaction
- still want to do operation? make a new transaction

unsure which? option 1: ask coordinator

- e.g. worker policy: keep asking if no outcome

unsure which? option 2: make sure coordinator resends outcome

- e.g. coordinator keeps sending outcome until it gets "yes, I got it" reply

two-phase commit: roles

typical two-phase commit implementation

several *workers*

one *coordinator*

- might be same machine as a worker

two-phase-commit messages

coordiantor → worker: PREPARE

> "will you agree to do this action?"
> on failure: can ask multiple times!

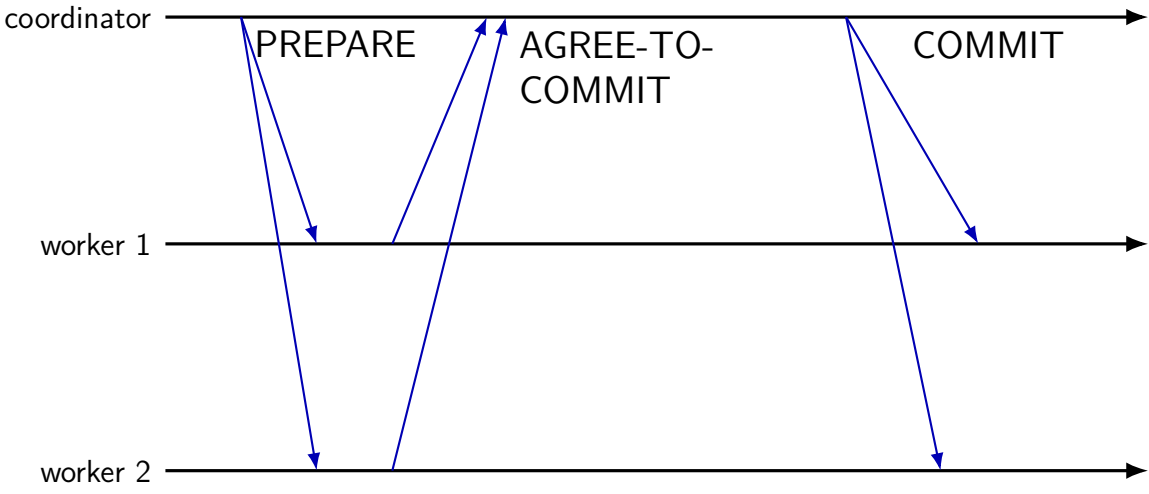
worker → coordinator:
AGREE-TO-COMMIT or AGREE-TO-ABORT

> worker records decision in log (before sending)

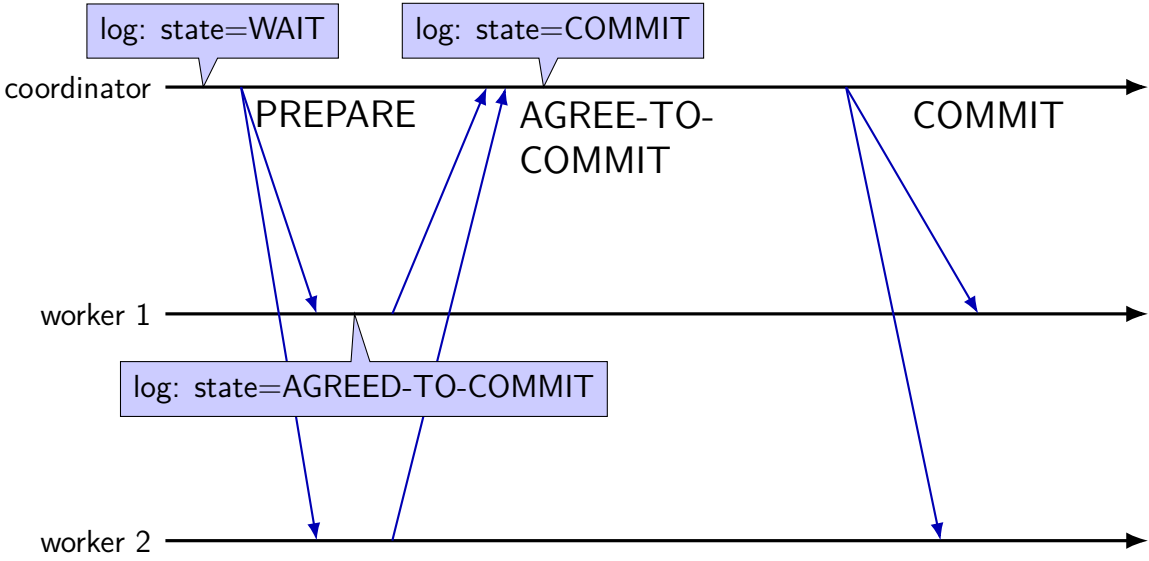
coordinator → worker: COMMIT or ABORT

> I counted the votes and the result is commit/abort
> only commit if all votes were commit

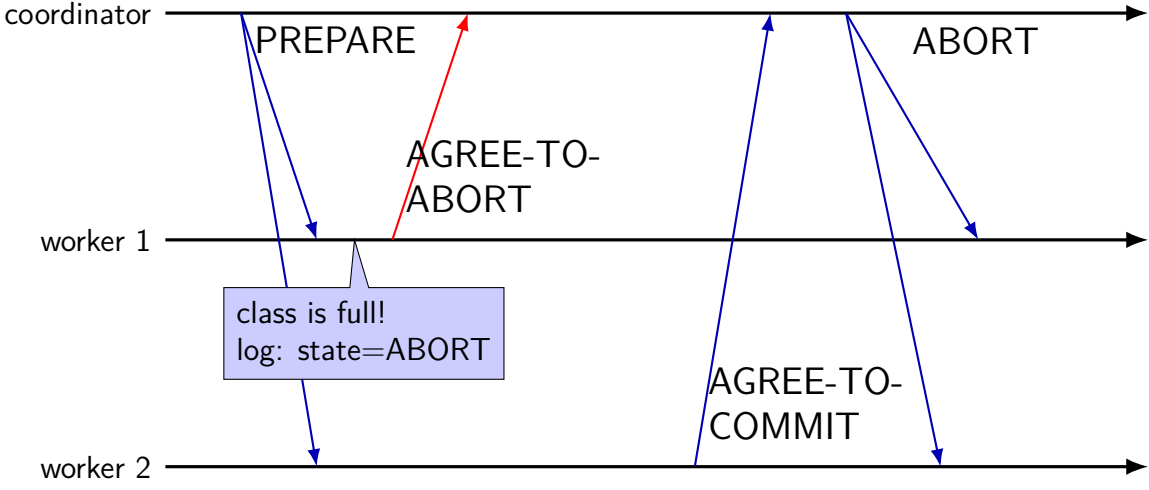
TPC: normal operation



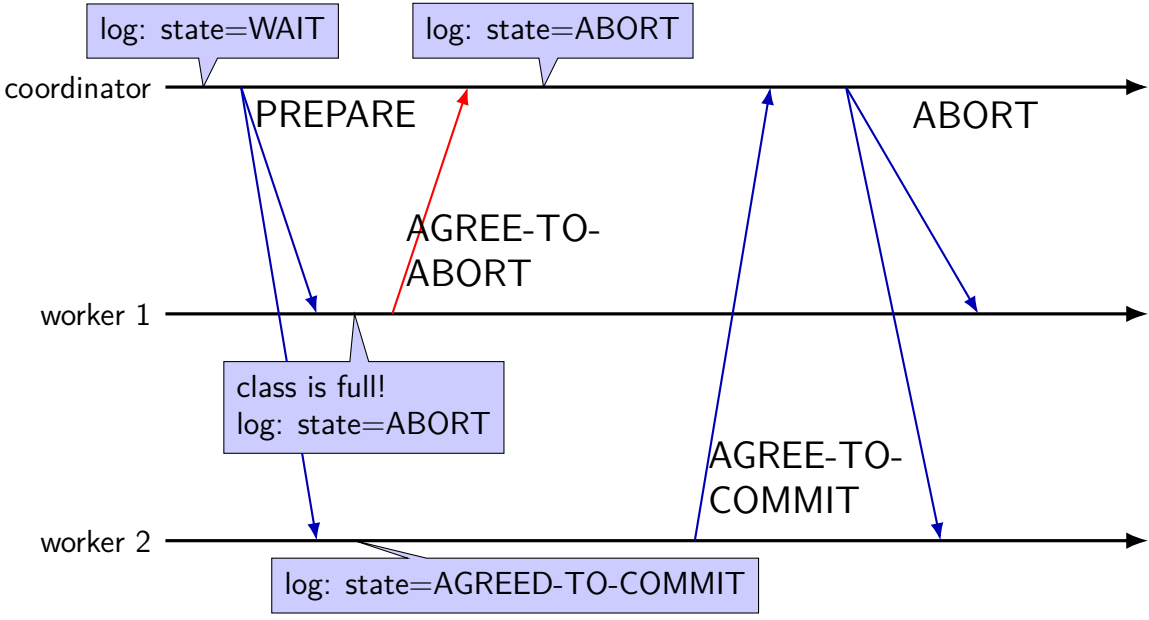
TPC: normal operation



TPC: normal operation — conflict



TPC: normal operation — conflict



exercise (1)

under what circumstances **may** a worker send vote to abort?

[A] in repsonse to a duplicate PREPARE message after replying to the first with a vote to commit

[B] after rebooting after a crash, if its log indicates it previously decided to vote to abort, but did not receive any decisions from the coordinator

[C] after rebooting after a crash, if its log indicates it previously decided to vote to commit, but did not receive any decisions from the coordinator

[D] after sending a vote to commit, but detecting that the coordinator crashed and has been down for a very long time

exercise (2)

under what circumstances **may** a coordinator send a decision to abort?

[A] when rebooting after a crash, after having last sent a request to vote to all but one worker and receiving votes to commit from all workers contacted

[B] when rebooting after a crash, when the log indicates that the last thing the coordinator did was deciding to commit but the log doesn't indicate that any workers were contacted

[C] after successfully sending a request for a vote to a worker, but not receiving the reply due to a network problem

two-phase commit: blocking

agree to commit "add student to class"?

can't allow conflicting actions...

...until know transaction *globally* committed/aborted

two-phase commit: blocking

agree to commit "add student to class"?

can't allow conflicting actions...

- adding student to conflicting class?
- removing student from the class?
- not leaving seat in class?

...until know transaction *globally* committed/aborted

waiting forever?

if machine goes away at wrong time, might *never* decide what happens

solution in practice: manual intervention

reasoning about protocols: state machines

very hard to reason about dist. protocol correctness

typical tool: state machine

each machine is in some state

know what every message does in this state

reasoning about protocols: state machines

very hard to reason about dist. protocol correctness

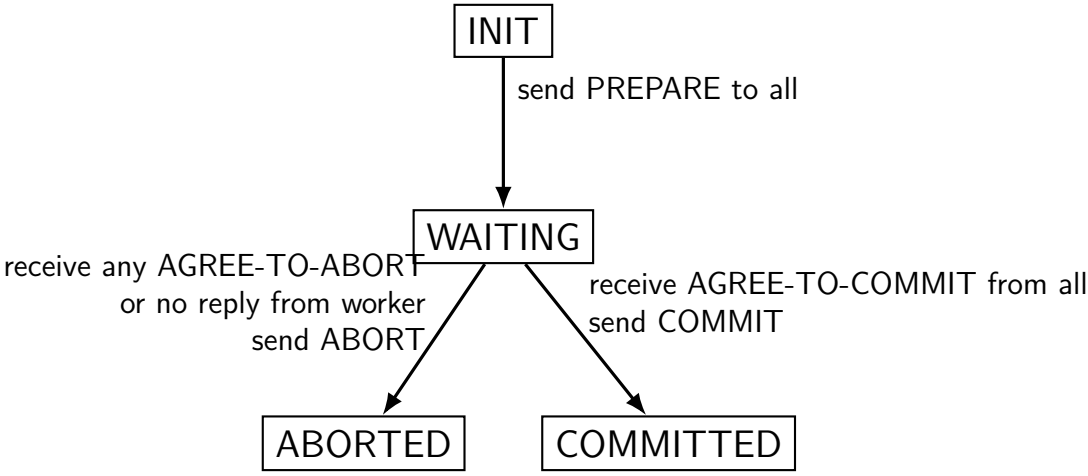
typical tool: state machine

each machine is in some state

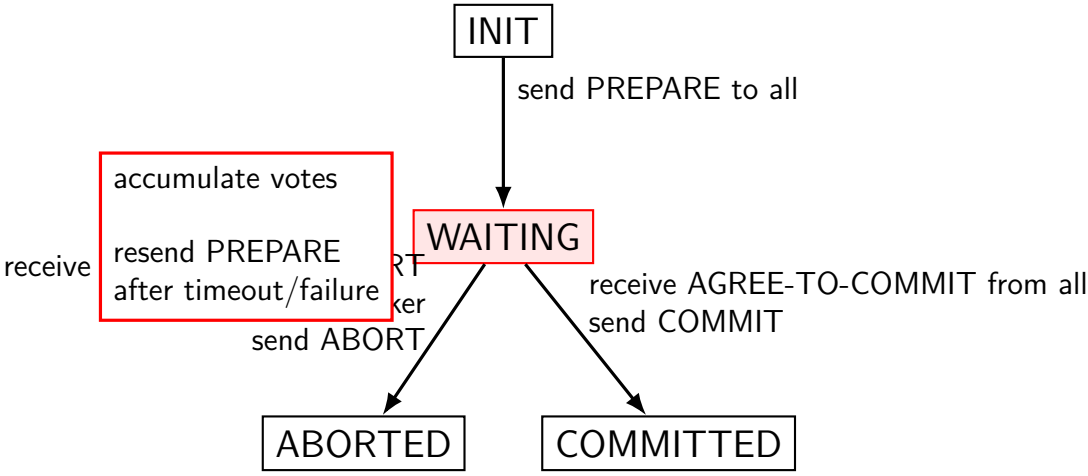
know what every message does in this state

avoids common problem: don't know what message does

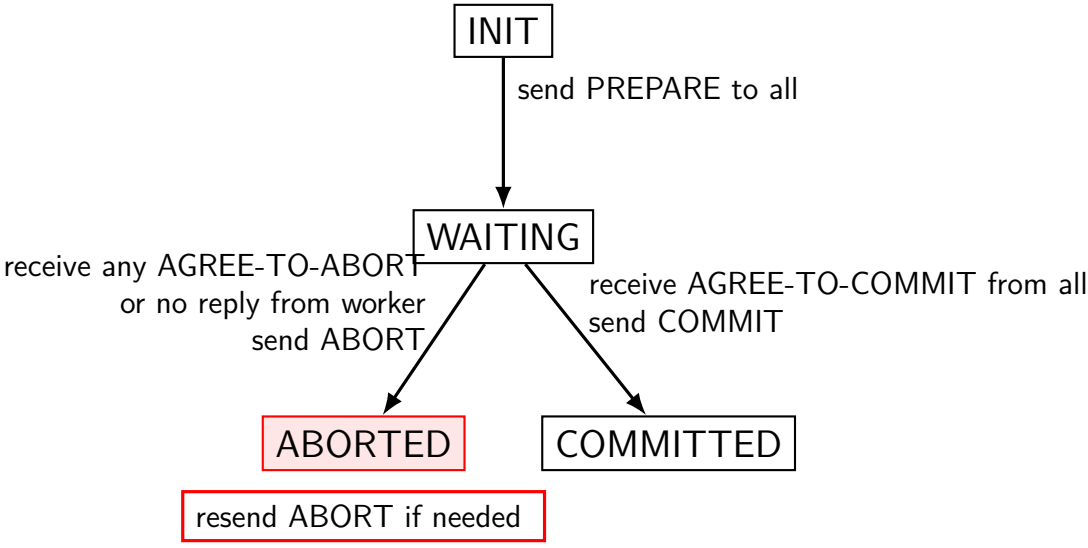
coordinator state machine (simplified?)



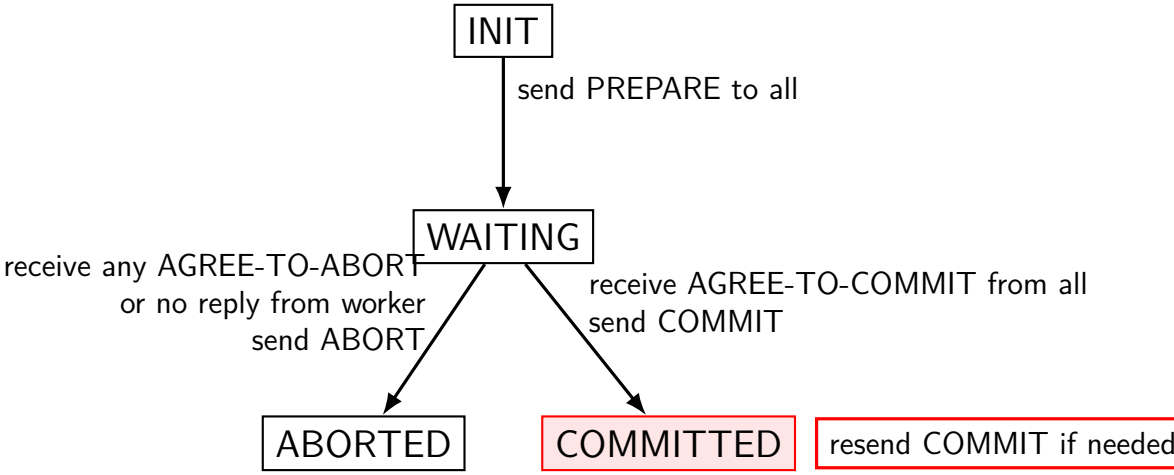
coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

worker doesn't get COMMIT/ABORT?

in assignment: worker sends acknowledgment; arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

worker doesn't get COMMIT/ABORT?

in assignment: worker sends acknowledgment; arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

workers need to handle duplicate messages!
coordinators need to handle duplicate replies!

W

in assignment: worker sends acknowledgment, arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

haven't sent commit? can abort instead (simpler?)

worker do

in assignment: worker sends acknowledgment, arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

- in assignment, errors detected only at coordinator
  using gRPC — so have return value from "COMMIT" RPC

worst case: log written, but message not sent

- in assignment, errors detected only at coordinator
  using gRPC — so have return value from "COMMIT" RPC

worker doesn't get COMMIT/ABORT?

in assignment: worker sends acknowledgment; arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

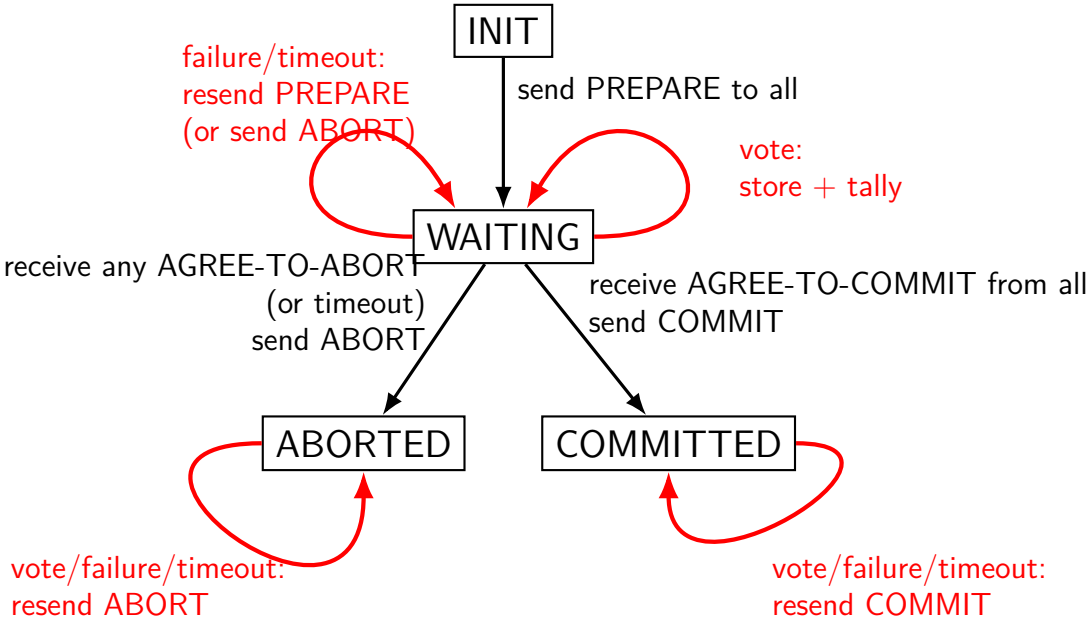
worst case: log written, but message not sent

normal strategy: wait for timeout, then resend
assignment: you throw exception; we'll restart (easier testing)

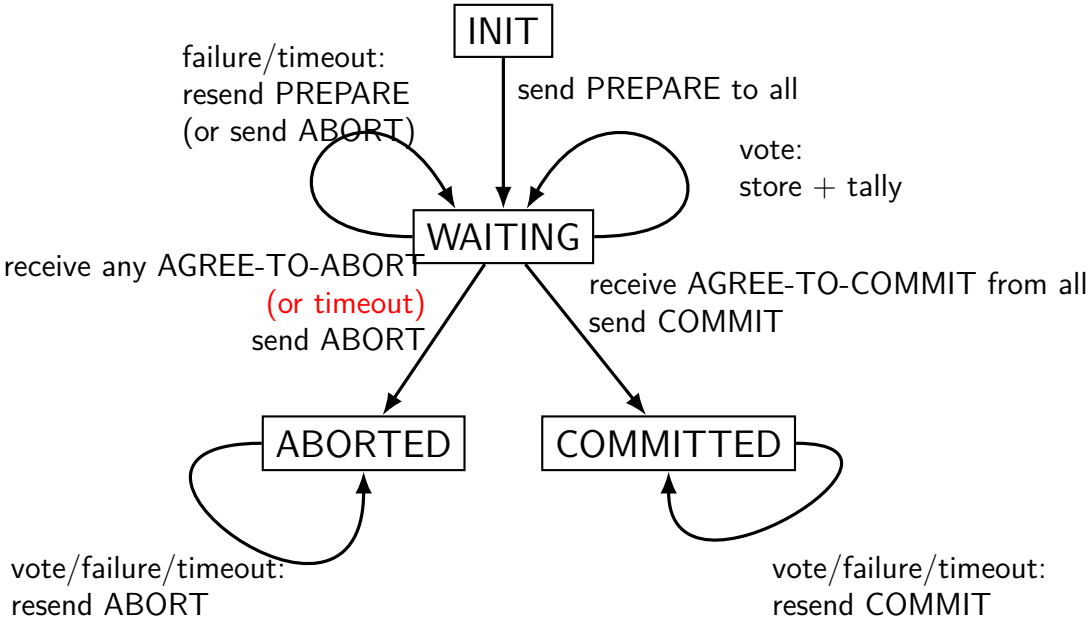
worker doesn't get COMMIT/ABORT?

- in assignment: worker sends acknowledgment; arrange retry if no ack
- other option: worker asks again after timeout

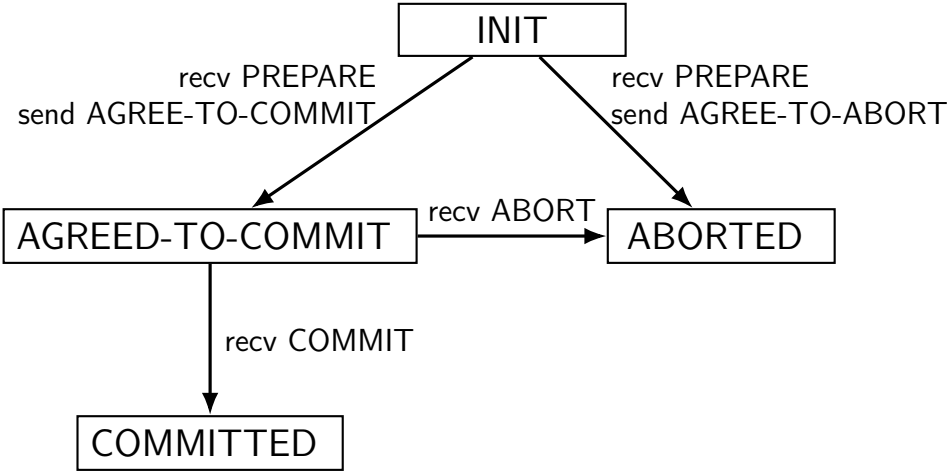
coordinator state machine (less simplified?)



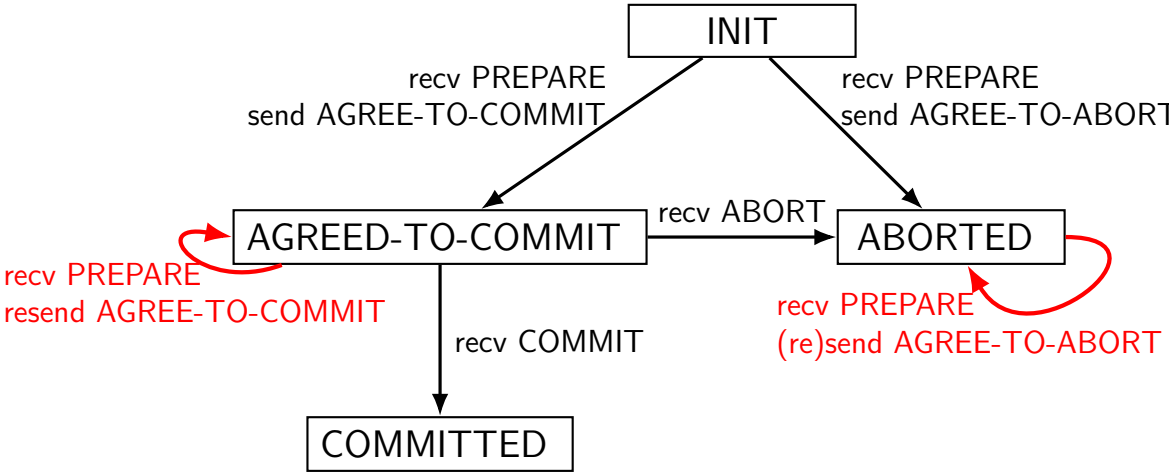
coordinator state machine (less simplified?)



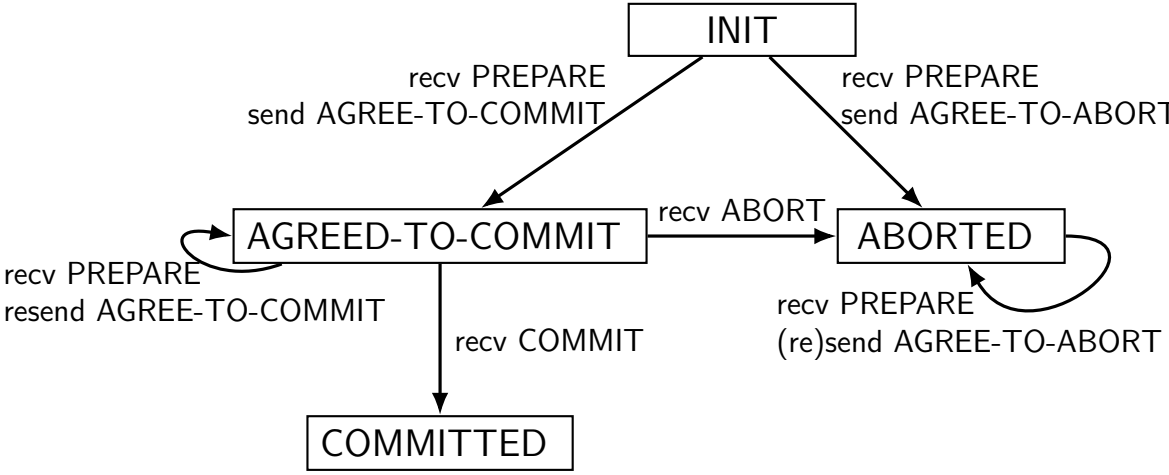
worker state machine (simplified)



worker state machine (less simplified?)



worker state machine (less simplified?)



worker failure recovery

worker crashes? *log* indicating last state

log written before acting on that state

if INIT:
wait for PREPARE (resent)?

if AGREE-TO-COMMIT or ABORTED:
resend AGREE-TO-COMMIT/ABORT

if COMMITTED:
redo operation (just like redo logging)

state machine missing details

really want to specify *result of/action for every message!*

worker recv ABORT in ABORTED: do nothing
worker recv ABORT in INIT: go to ABORTED
worker recv PREPARE in COMMITTED: ignore?
...

everything specified: machine checkable?

want to discard finished transactions eventually

slides we didn't get to (will cover next Tues)

two-phase commit assignment

two phase commit assignment

store *single value* across workers

single coordinator sends messages to/from workers to change values

- workers current value can be queried directly

goal: several replicas all have same value *or unavailable*

...even if failures

assignment: RPC

coordinator talks to worker by making RPC calls

workers only talk to coordinator by replying to RPC

- example: make "prepare" call, worker's "agree-to-X" is return value

RPC system detects worker being down, network errors, etc.

- become Python exception in coordinator

coordinator verifies Commit/Abort received instead of worker asking again

- automatic: Commit/Abort message is RPC call with return value; RPC call fails if problem getting return value

workers might never agree-to-abort (and that's okay)

- no conflicting operations: only crash or agree-to-commit

assignment: failure recovery

to simplify assignment: always return error if you detect failure

assume testing code/user will restart the coordinator+workers

coordinator sends messages to workers on reboot to recover

- resend prepare or commit, abort, etc.

assignment: failure types

send RPC and

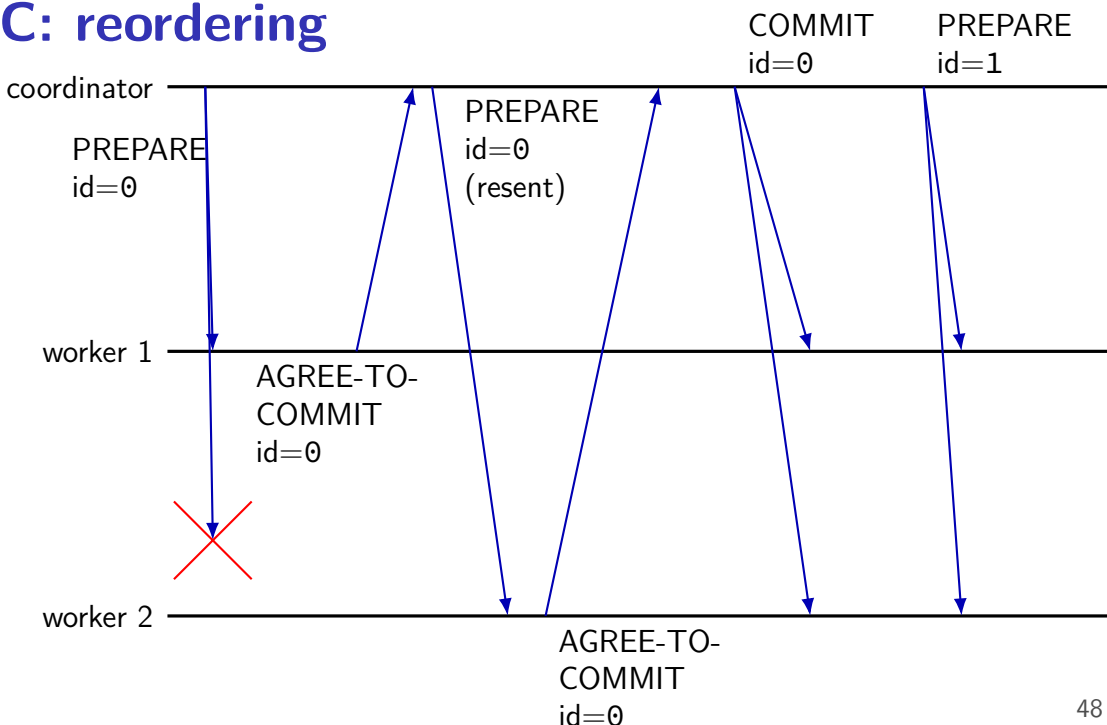
- it gets lost
- it gets sent, but acknowledgment/reply is lost
- it gets sent, but delayed until after another RPC

assignment: failure types

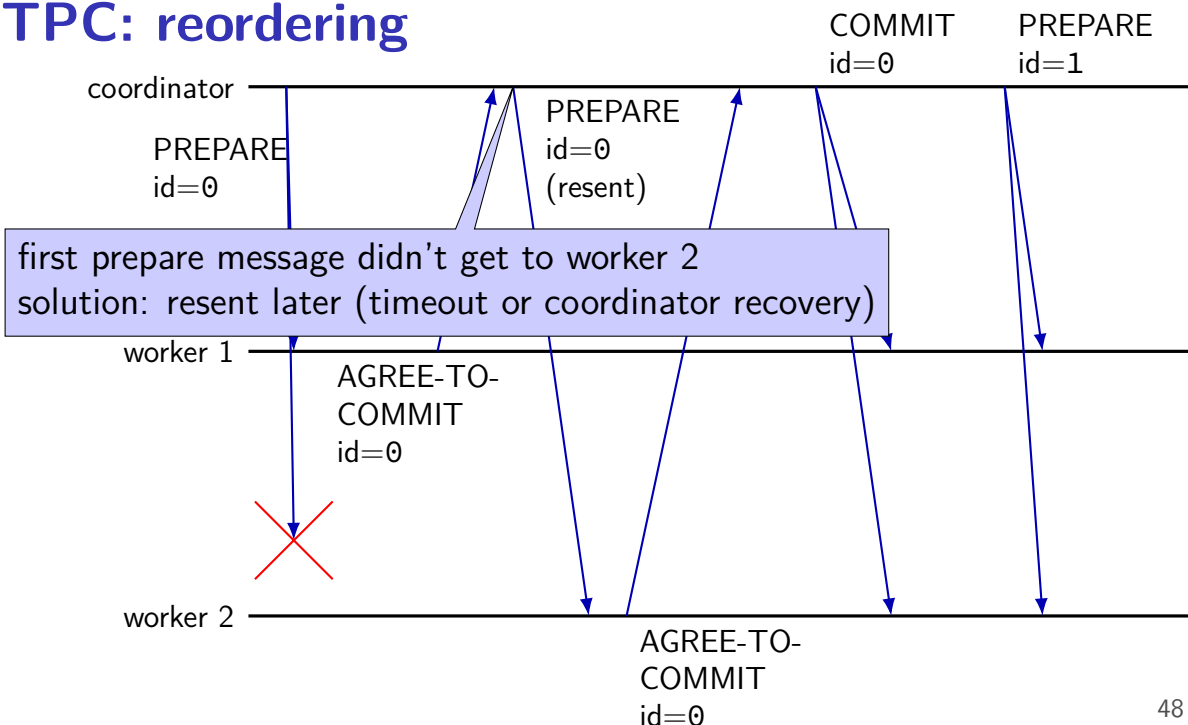
send RPC and

- it gets lost
- it gets sent, but acknowledgment/reply is lost
- it gets sent, but delayed until after another RPC

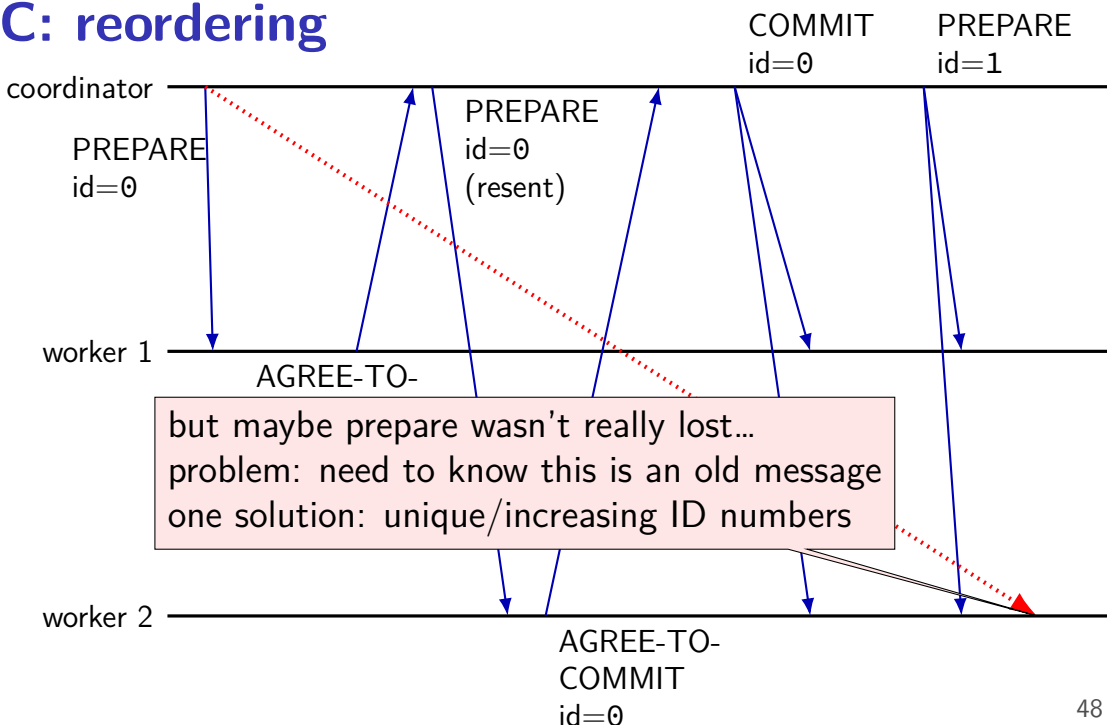
TPC: reordering



TPC: reordering



TPC: reordering



message reordering and assignment

assignment: you need to worry about reordering

- connections prevent reordering, but...
- RPC system doesn't prevent it: can use multiple connections

problem: old request *seems to fail*, but is actually slow

you repeat old request again

later on slow old request reaches machine $\rightarrow$ must be ignored!

solution: sequence numbers or transactions ID and/or timestamps

- some way to tell "this is old"

worker failure during prepare

worker failure after prepare without sending vote?

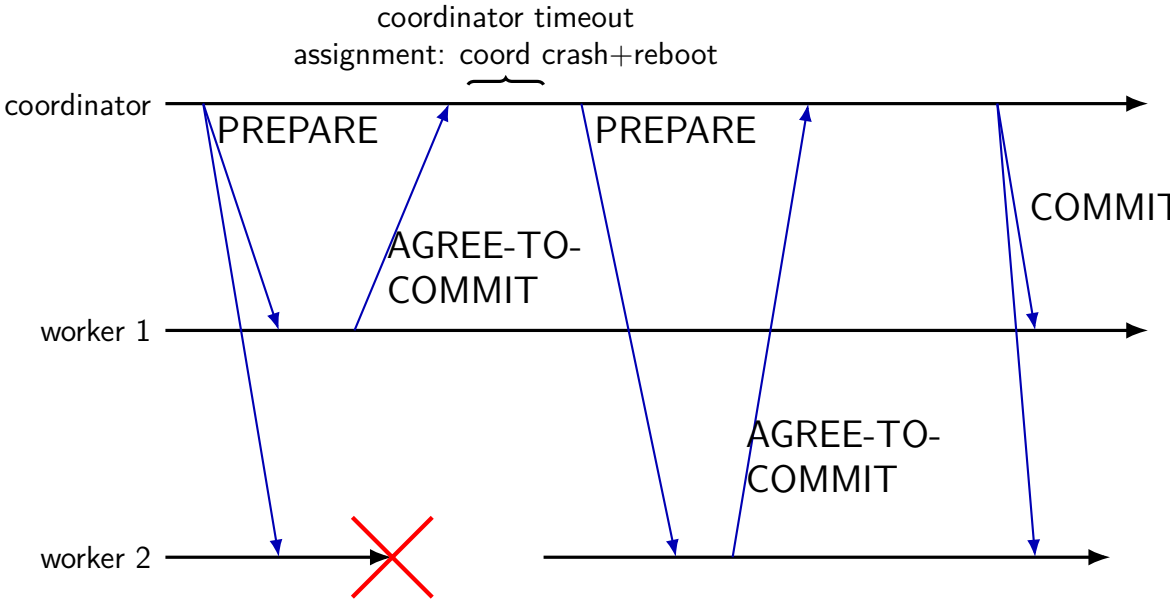
- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

worker failure during prepare

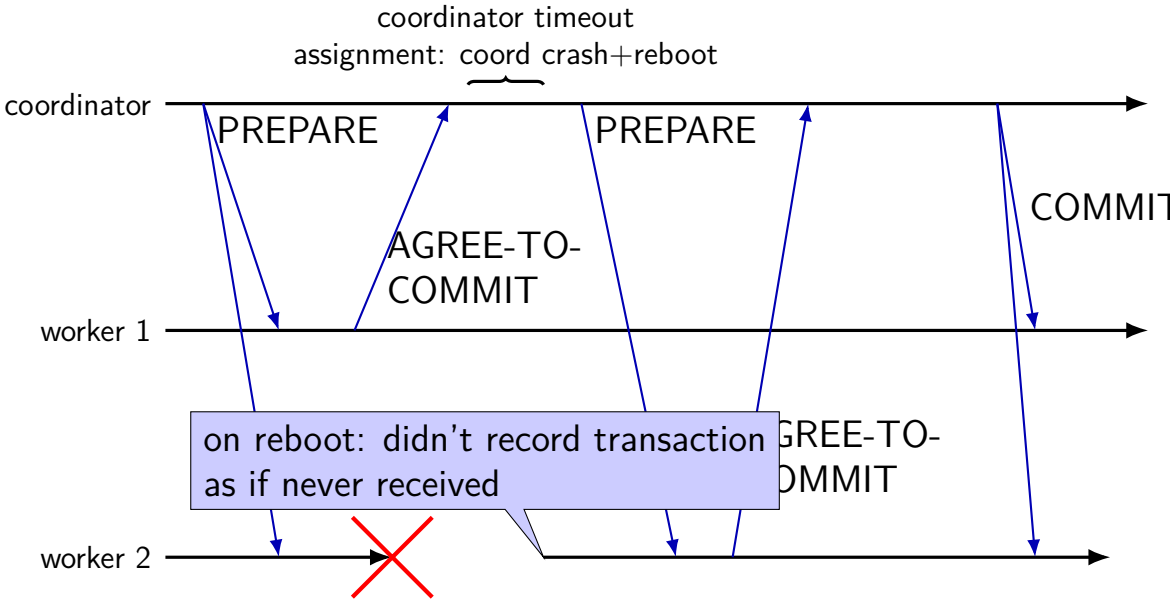
worker failure after prepare without sending vote?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

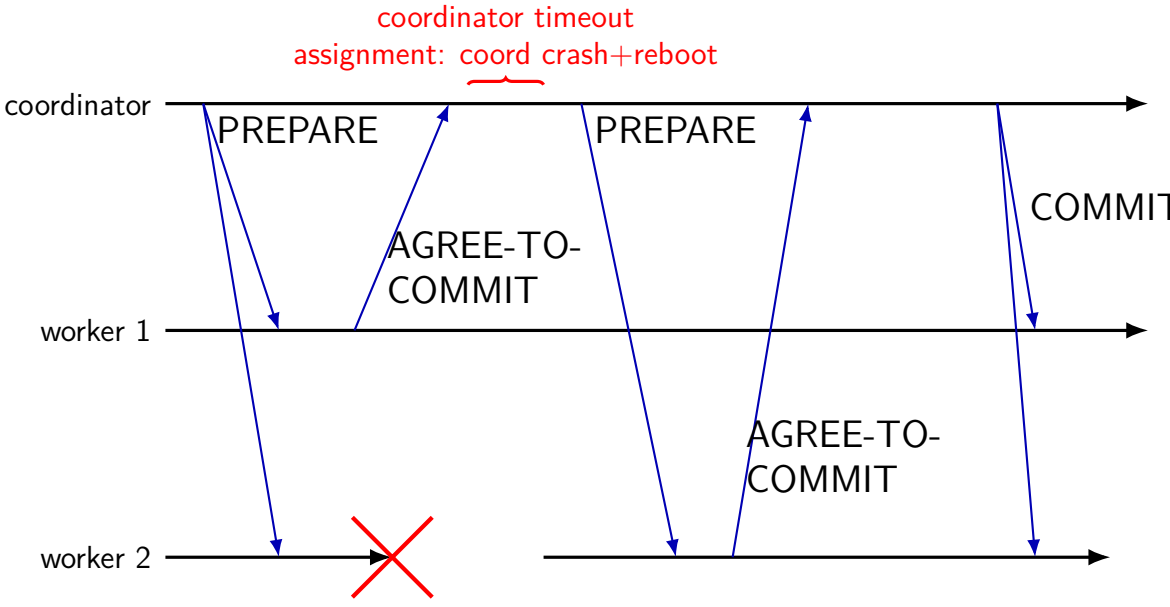
TPC: worker fails after prepare (1a)



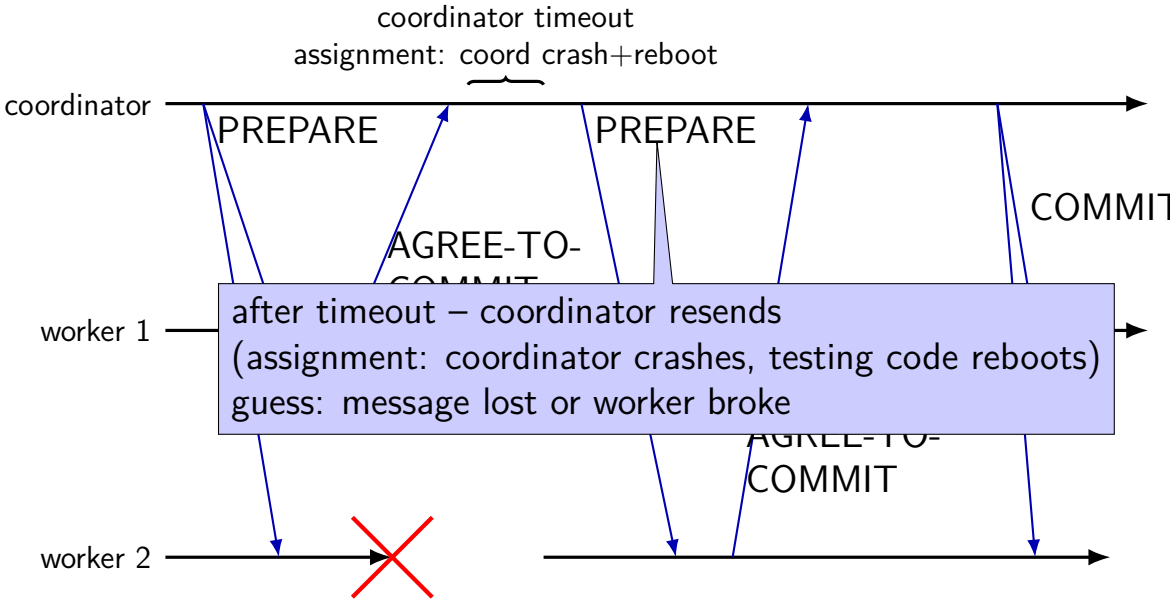
TPC: worker fails after prepare (1a)



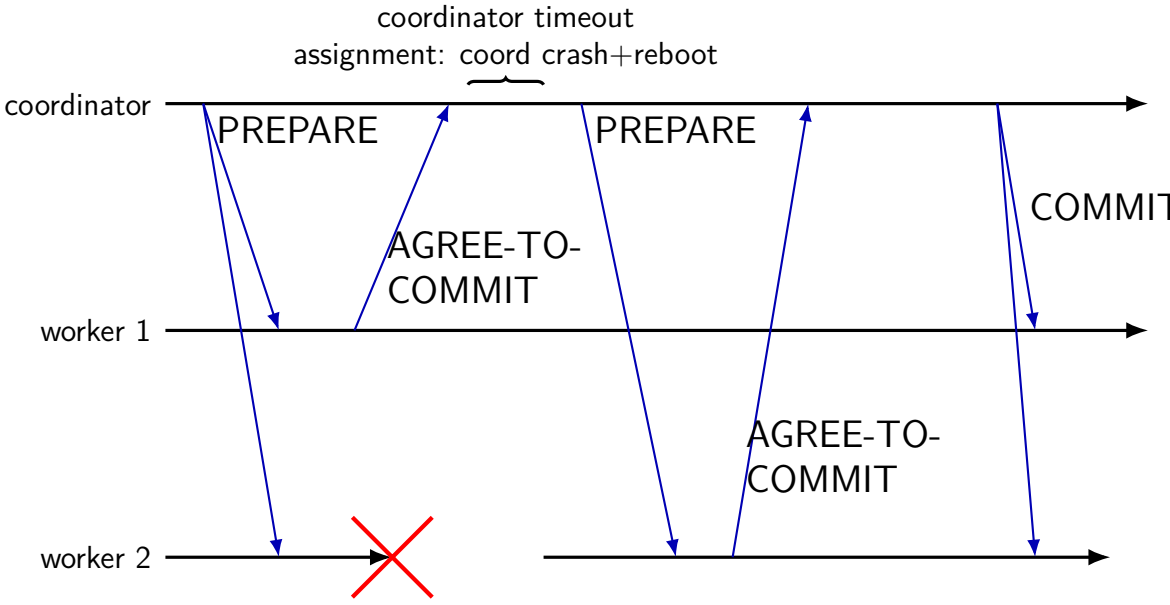
TPC: worker fails after prepare (1a)



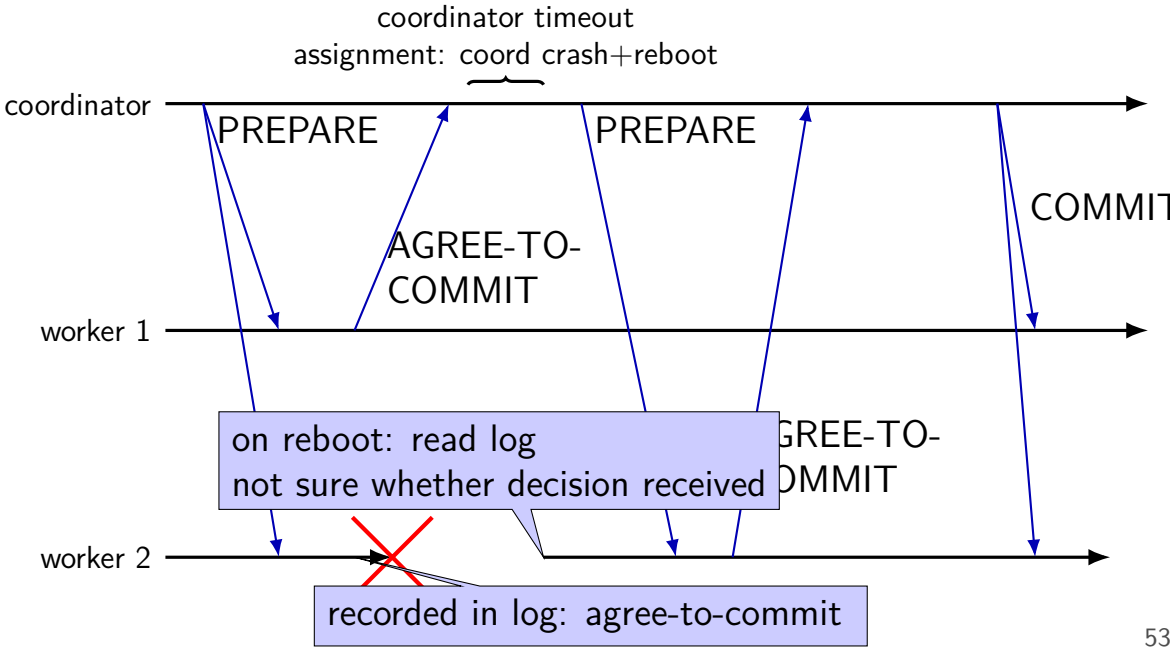
TPC: worker fails after prepare (1a)



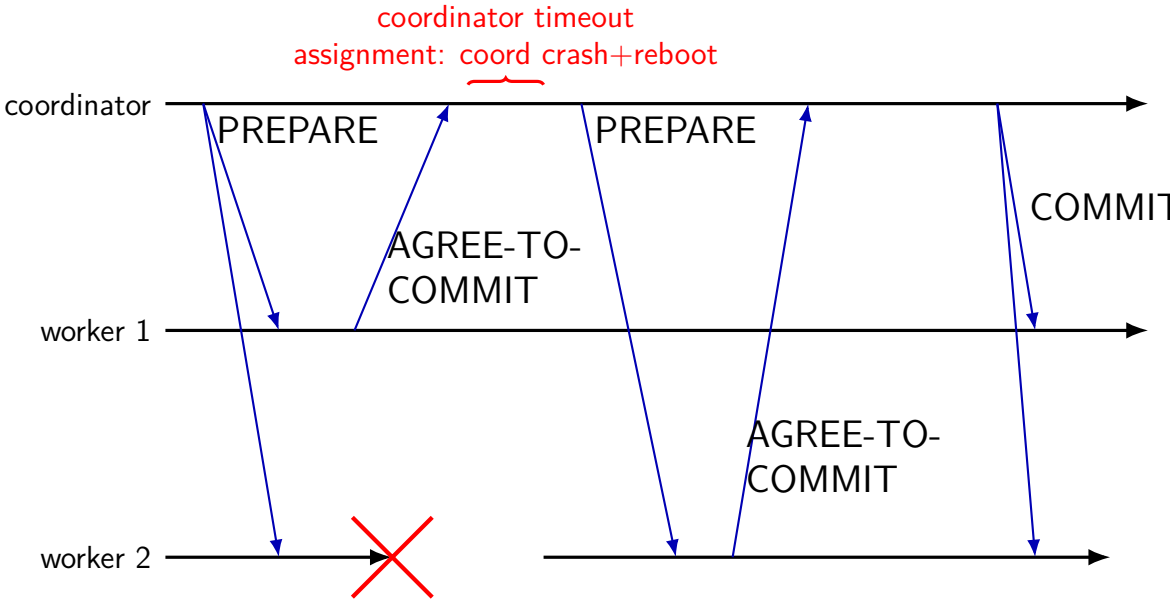
TPC: worker fails after prepare (1b)



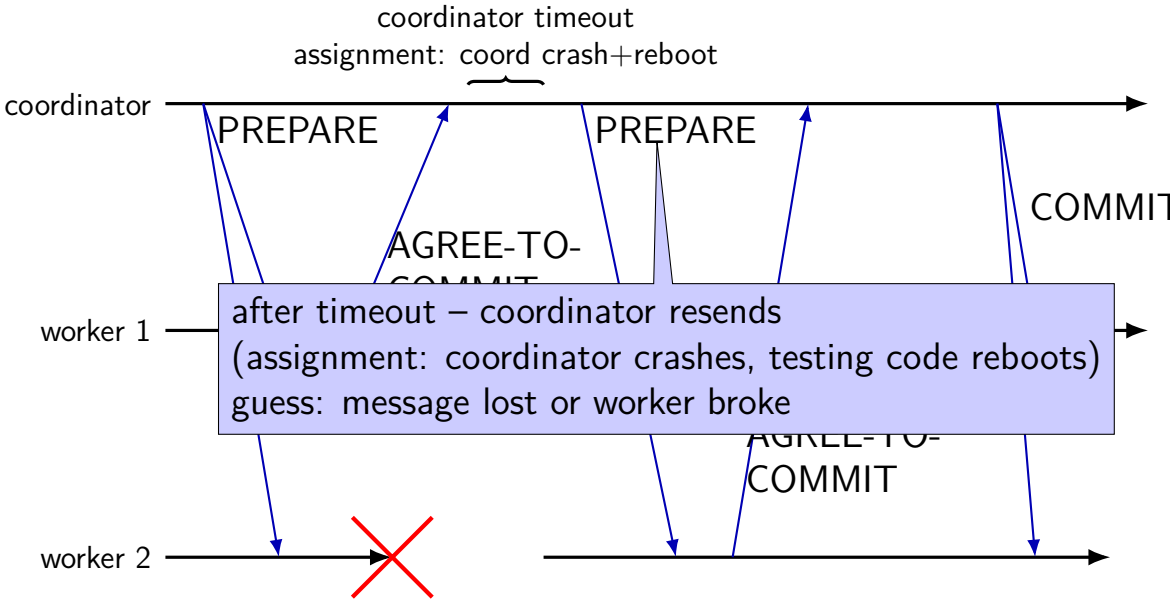
TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)

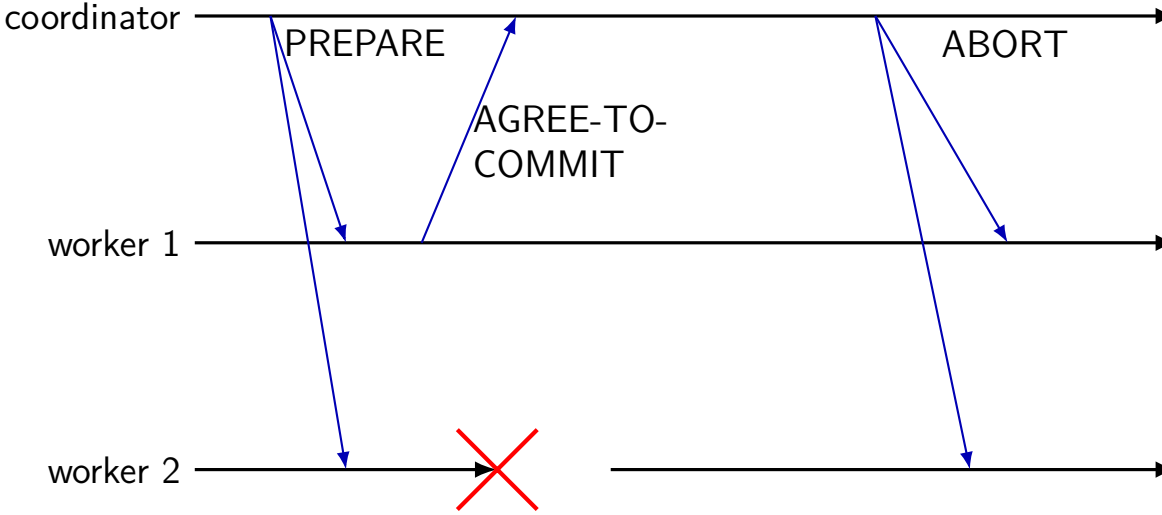


worker failure during prepare

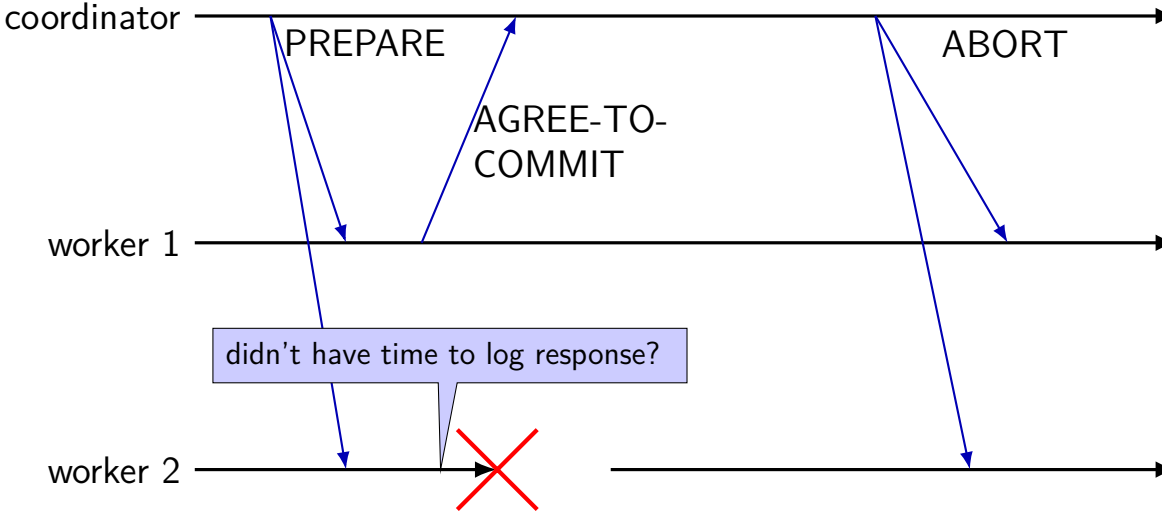
worker failure after prepare without sending vote?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

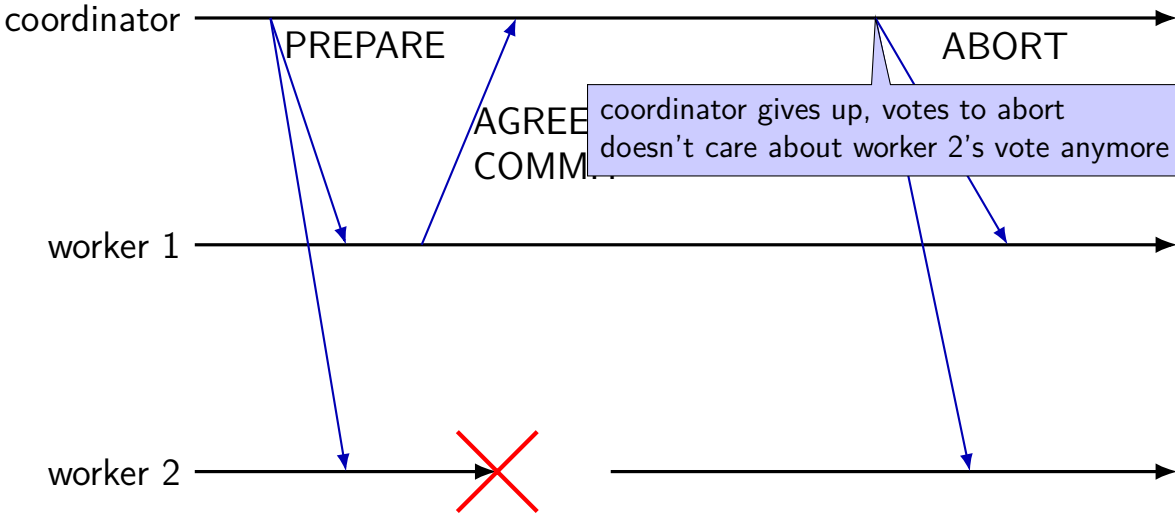
TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)

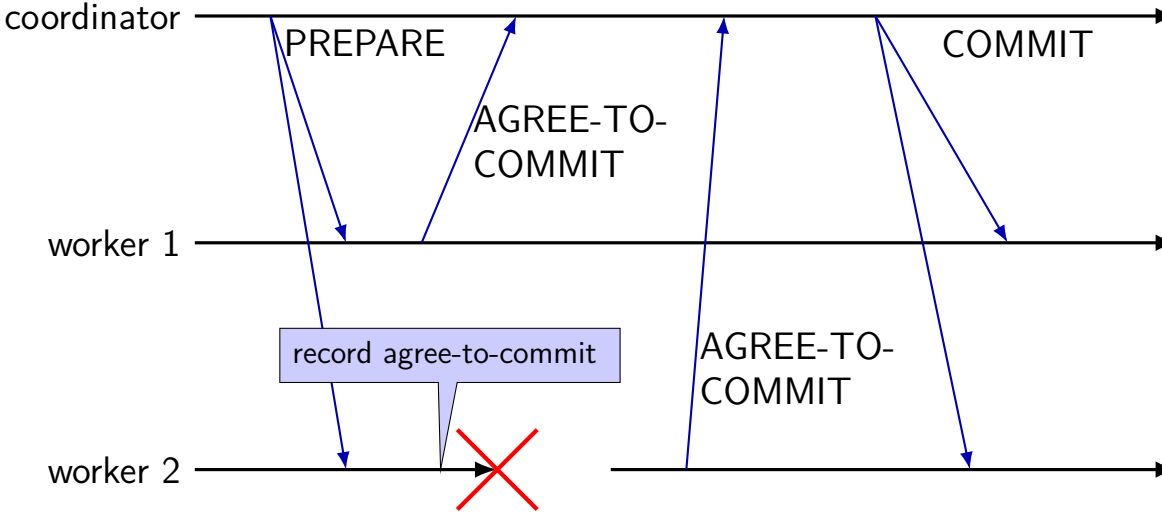


worker failure during prepare

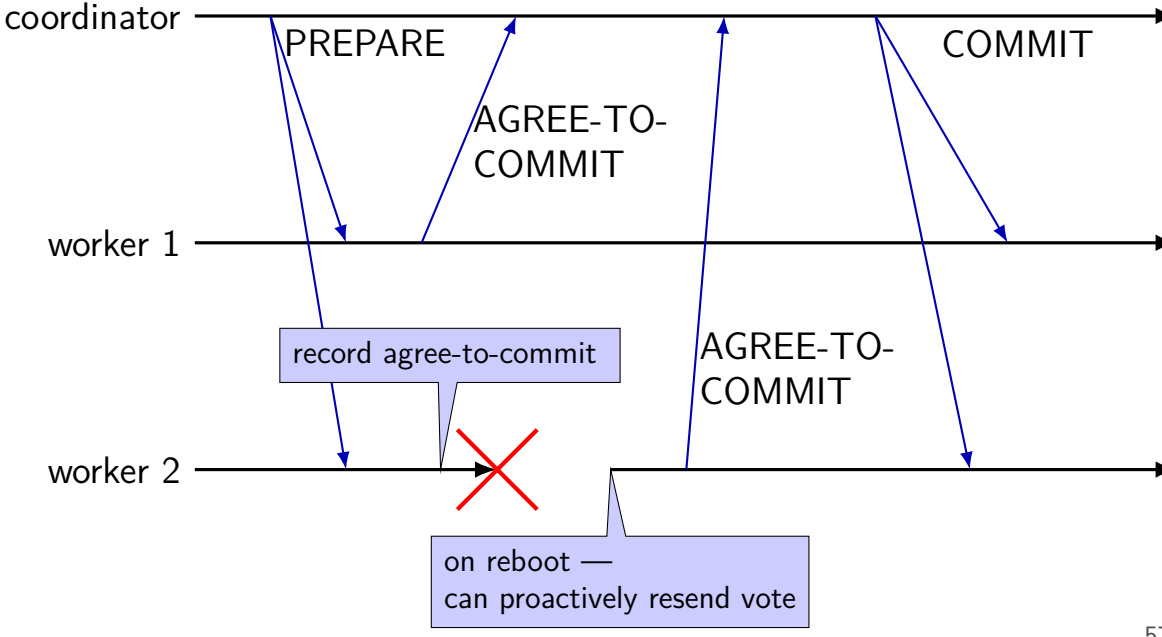
worker failure after prepare without sending vote?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

TPC: worker fails after prepare (3)



TPC: worker fails after prepare (3)



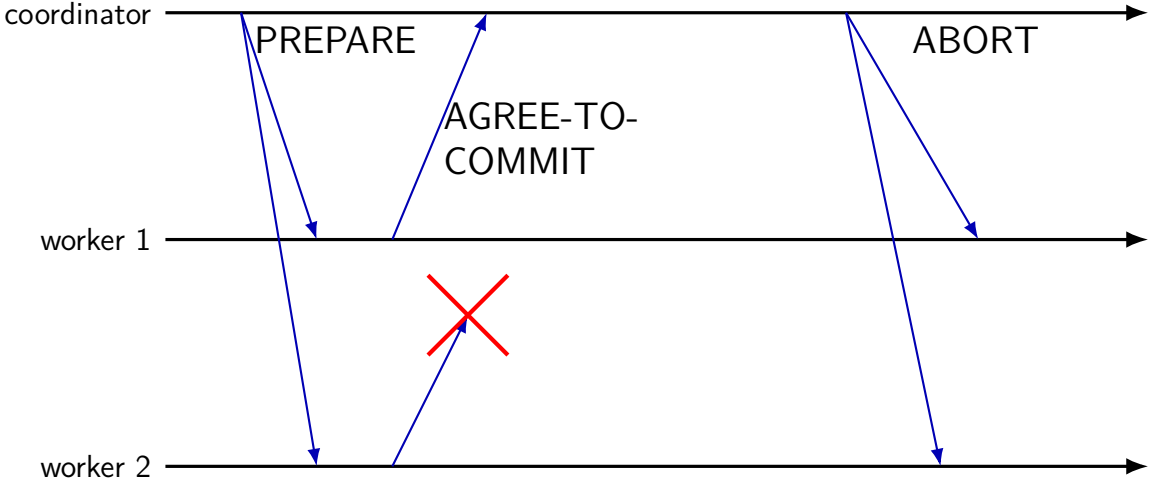
network failure after during voting?

network failure during voting $\approx$ node failure

same options:

- coordinator resends PREPARE
- coordinator gives up
- worker resends vote

TPC: network failure (1)



worker failure during commit

worker failure during commit?

- option 1: coordinator resends outcome somehow?
  - requires acknowledgements from worker
  - required for assignment
- option 2: worker resends vote (coordinator resends outcome)

NB: coordinator cannot give up

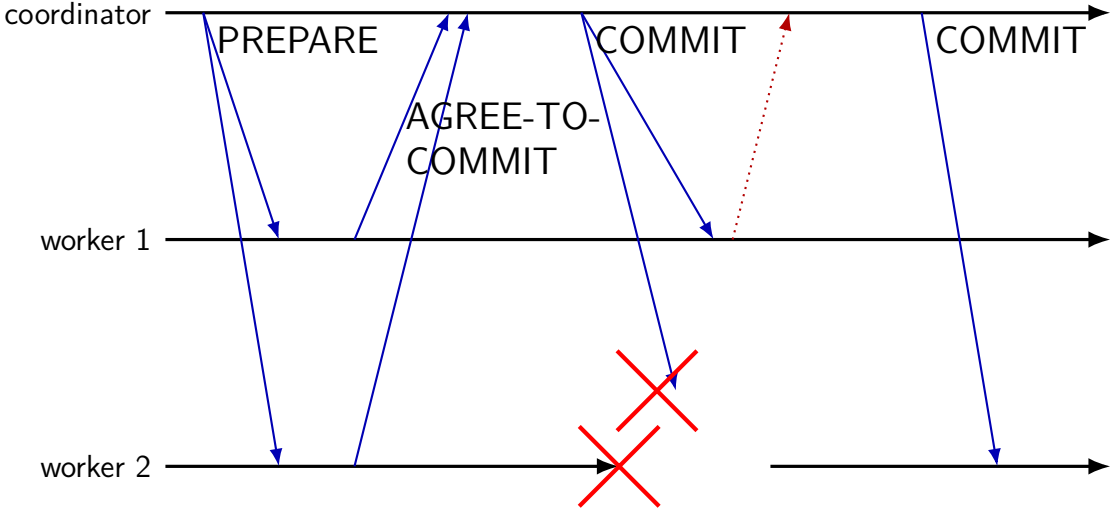
worker failure during commit

worker failure during commit?

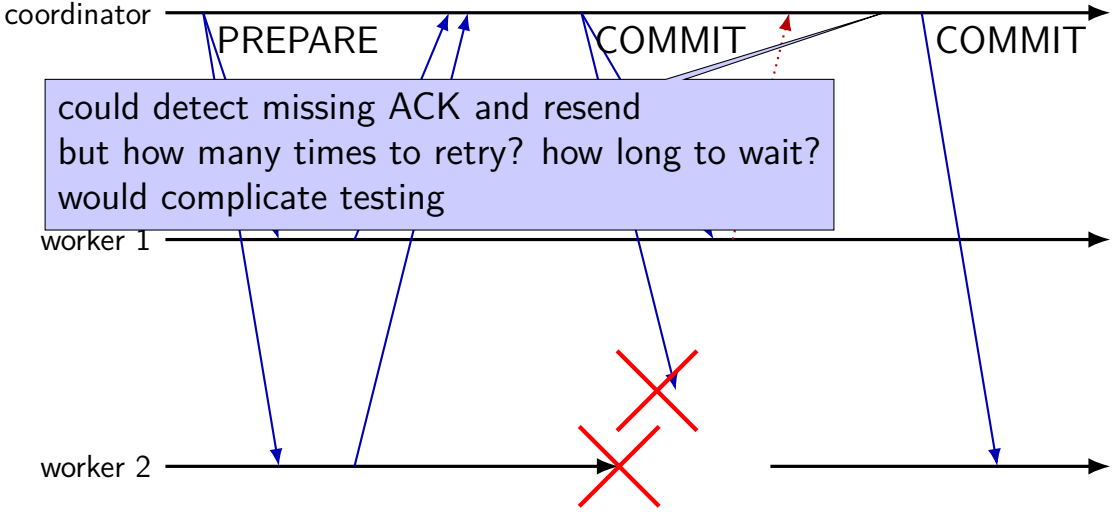
- option 1: coordinator resends outcome somehow?
  - requires acknowledgements from worker
  - required for assignment
- option 2: worker resends vote (coordinator resends outcome)

NB: coordinator cannot give up

coordinator resend automatically



coordinator resend automatically



twophase assignment recovery

on failure: we'll restart everything that failed

"crash-oriented computing": simplifies implementation

- you need to handle everything crashing anyways...
- so just make that the only way you handle errors

backup slides

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

other model: every node has a copy of data

goal: work (including updates!) despite a few failing nodes

just require "enough" nodes to be working

for now — assume fail-stop

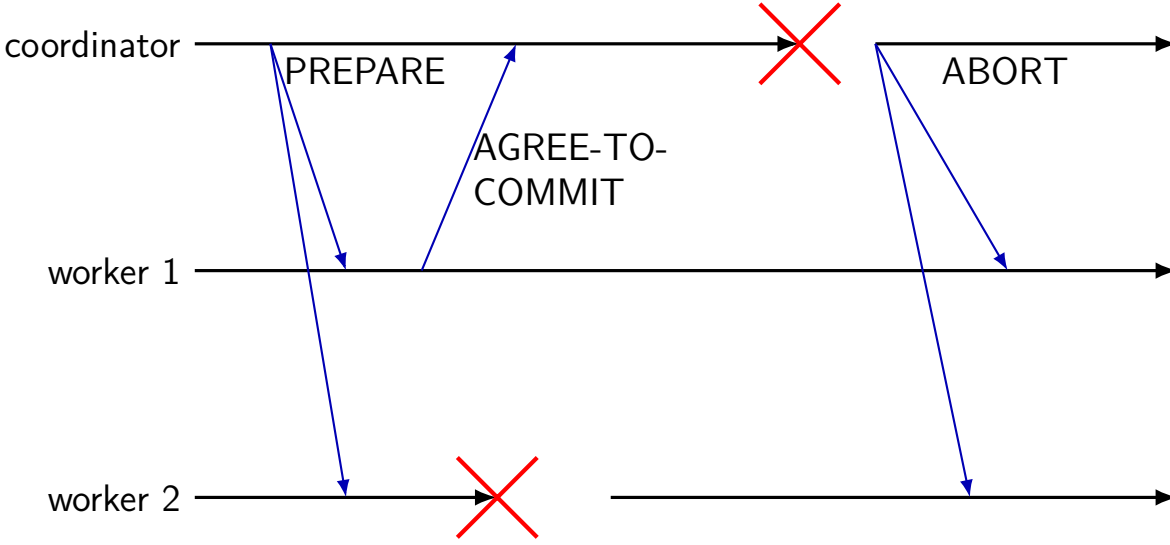
nodes don't respond or tell you if broken

assignment: failure types

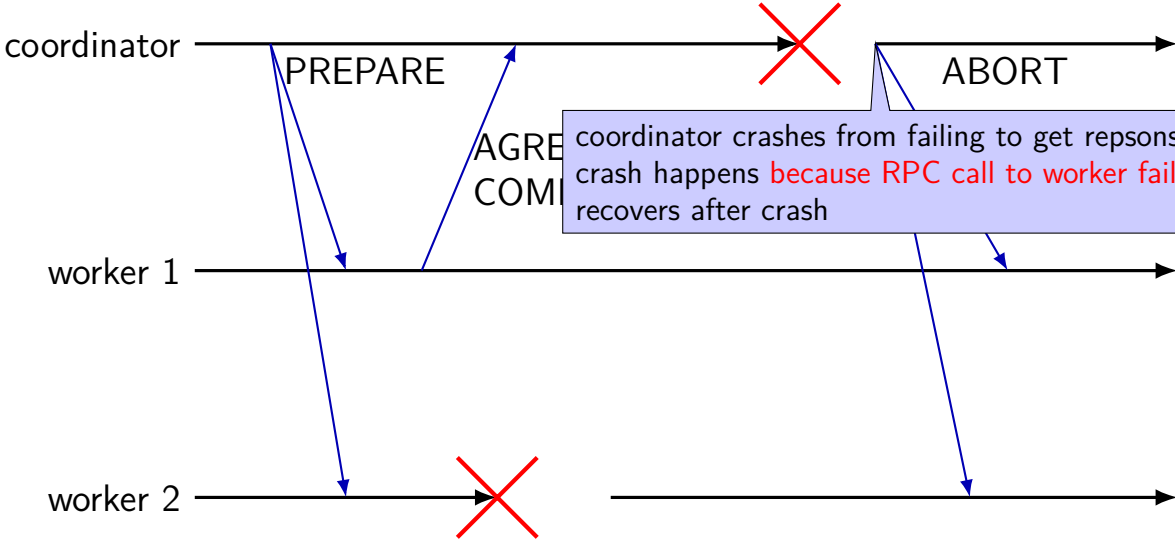
send RPC and

- it gets lost
- it gets sent, but acknowledgment/reply is lost
- it gets sent, but delayed until after another RPC

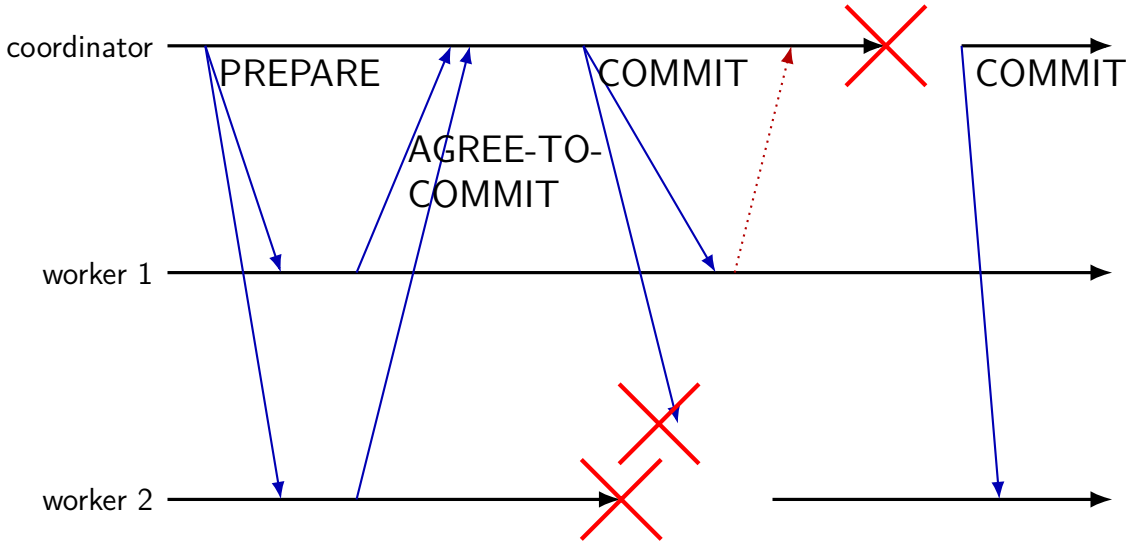
assignment: fails during prepare



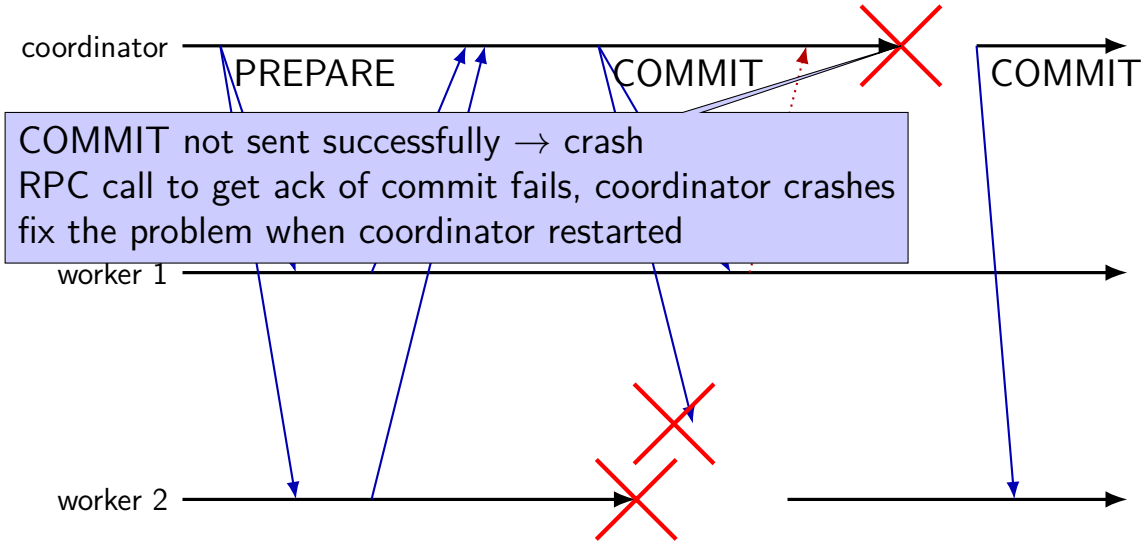
assignment: fails during prepare



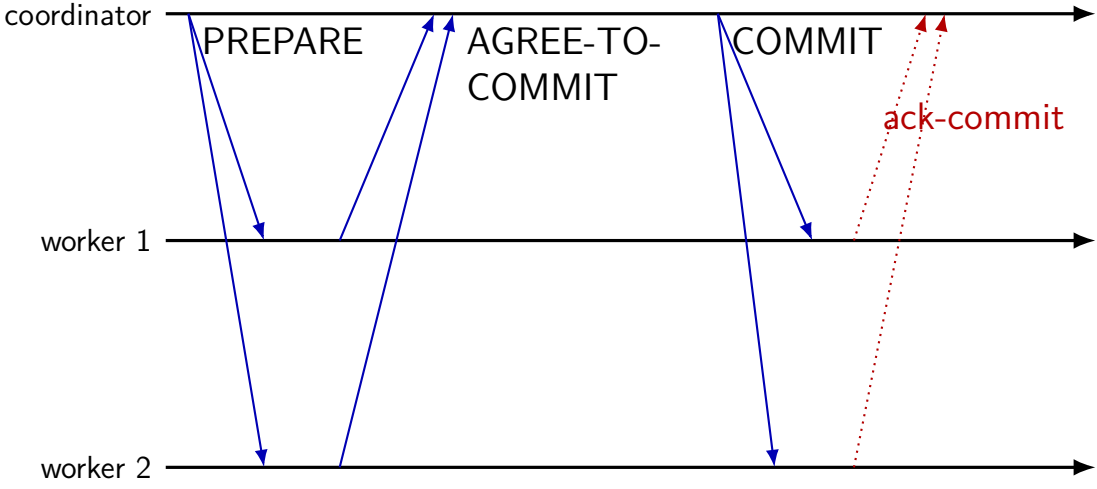
assignment: failuring during commit



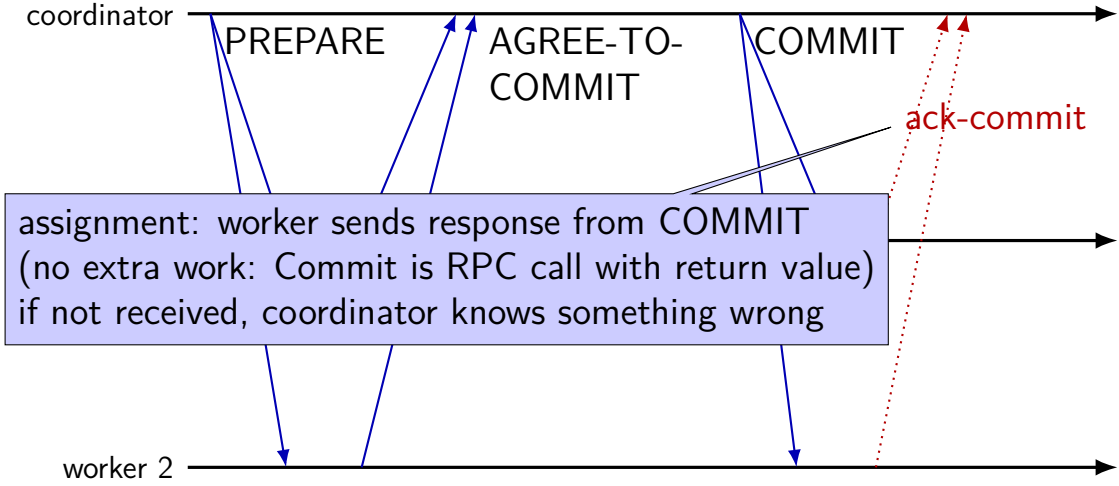
assignment: failuring during commit



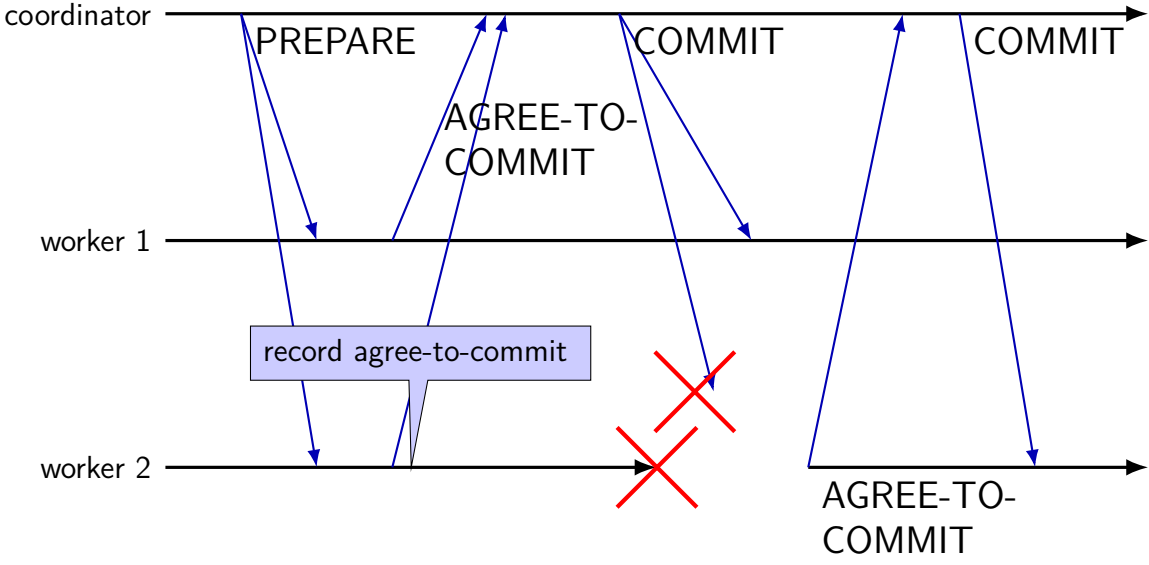
aside: worker ACKs



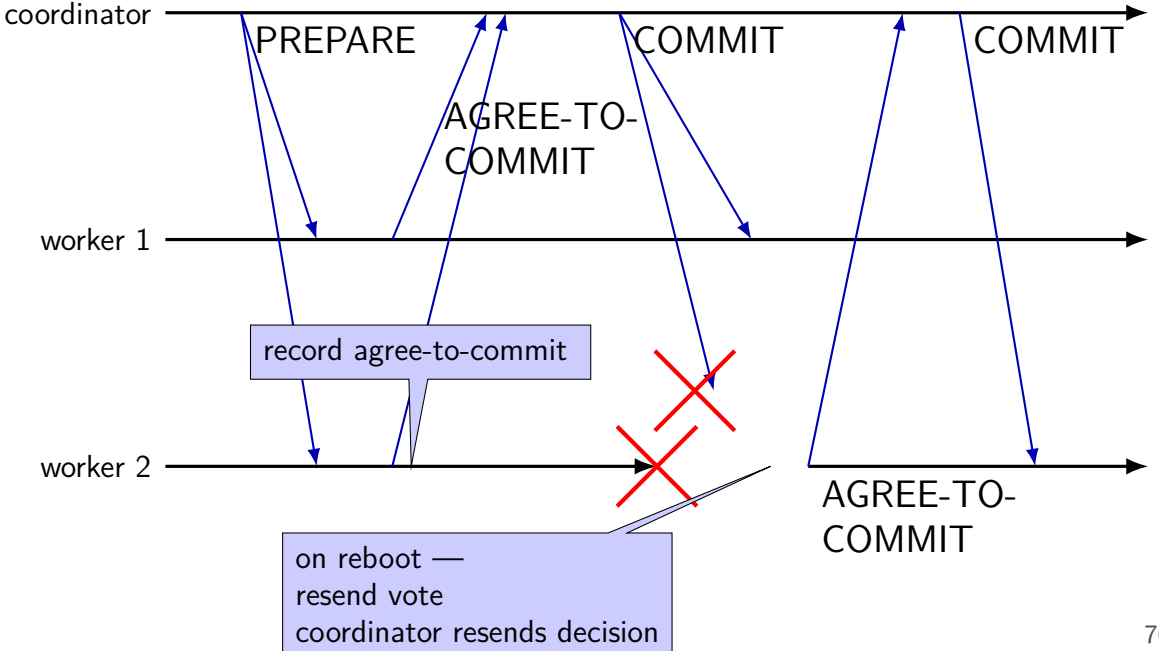
aside: worker ACKs



TPC: worker revoting



TPC: worker revoting

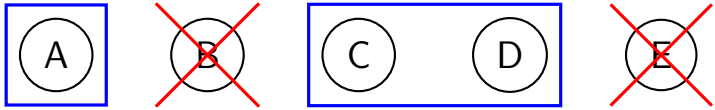


quorums (1)

perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

quorums (1)



perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

if A, C, D agree: that's enough

B, E will figure out what happened when they come back up

quorums (2)

requirement: quorums overlap

overlap = *someone in quorum* knows about every update

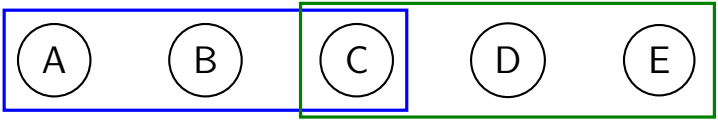
e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

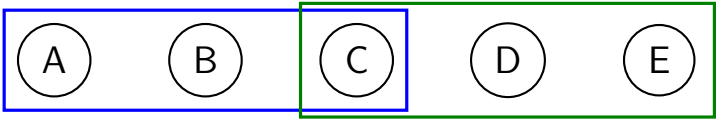
e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates